Proceedings of the 8th International Conference on Applied Informatics
Eger, Hungary, January 27–30, 2010. Vol. 2. pp. 409–416.

A Small Survey of Java Specification Languages*

Gábor Kusper[a], Gergely Kovásznai[a], Wolfgang Schreiner[b], Gábor Guta[b], János Sztrik[a]

[a]Eszterházy Károly College, Mathematics and Informatics Institute

[b]Johannes Kepler University Linz, RISC Institute

Abstract

This work is motivated by the final goal to formalize design patterns and design principles. A step towards this goal was to survey state-of-the-art Java specification languages. In a specification language one can describe properties of a program. The specification can involve some related runtime states. Some specification languages can specify properties about only a single state, others about a pair of states, but the most general ones are designed to speak about sequences of states. The specification can involve also constraints on the structure of the program. Some specification languages can speak about only the static structure of the source code, others also about the dynamic structure of runtime data. All languages have some tools to check the specifications, either statically or dynamically. We classify the Java specification languages and their tools along these dimensions, i.e., the number of related runtime states in the specification, the way to access program structure in the specification, and the way to verify the specifications. We found that most of the specification languages cannot directly query the structure of the program, but they can use reflection to do this. We give an example how to use reflection to specify that a class is semi-final, i.e., we can inherit a new class from it but this class may not have public methods. This means that the interface of the classes in the hierarchy is finalized. This property supports the "program to an interface, not an implementation" design principle.

Keywords: design by contract, behavior specification languages.

MSC: 68N19 Object-oriented programming.

*Supported by the Austrian Academic Exchange Service (ÖAD) under the contract HU 14/2009.

1. Introduction

We survey Java specification languages to present the state of the art in specification languages and verification tools. In a specification language one can describes properties of a program. The specification can involve some related runtime states. Some specification languages can specify properties about only a single state, others about a pair of states, but the most general ones are designed to speak about sequences of states. The specification can involve also constraints on the structure of the program. Some specification languages can speak about only the static structure, others also about the dynamic one. All languages have some tools to check the specifications. This verification is performed either statically at compile-time or dynamically at run-time.

We classify the Java specification languages and their tools along these dimensions:

- the number of related runtime states in the specification,
- the way to access program structure in the specification, and
- the way to verify the specifications.

The goal of this paper is not a complete survey on Java specification languages, but to give the dimensions needed to write a complete survey.

We found that most of the specification languages cannot directly query the structure of the program, but they can use reflection to do this. As an example, we show how to use reflection to specify that a class is semi-final, i.e., we can inherit a new class from it but this new class may not have additional public methods. This means that the interface of the classes in the hierarchy is finalized. This property supports the "program to an interface, not an implementation" design principle.

2. Java Specification Languages and Tools

In this section we give a very short overview about the most well known Java specification languages and their tools. In the specification we can describe how the program supposed to work, in other words, how should it behave. From this comes the name behavioral specification languages. Specification languages are motivated by the design by contract concept. Here we think on a method in terms of its contract, the contract between the caller and the called method. The contract says that if the caller complies with the precondition of the method, then the postcondition will be true after the method call. This means that in the design by contract concept each method has a pre- and a postcondition. Specification languages are used to write these conditions.

The advantages of using specification languages are that

- we can check if the actual code fulfills the specification or not,

- the pre- and postcondition of a method can be stated clearly in the program, not only in some vague way in the documentation,
- in an abstract class not only the signatures of the methods can be specified for the extending classes but also the expected behavior.

The disadvantages of using specification languages are that

- some aspects of the program have to be written twice, once in the specification and once in the source code, and one has to change both if one of them changes,
- average programmers are not enough well trained to write complicated specifications,
- current tools are not enough user friendly.

Java assert. The most basic way to introduce specification is to use the built-in Java "assert" statement. It has two kinds of syntax:

```
assert test;
assert test : ErrorMessage;
```

The test is a boolean formula. If it is true, then "assert" does not do anything. If it is false, then an AssertionError is thrown with no detail message. In the second form we can also give the ErrorMessage which is a string. If the test contains a method call which changes the inner state, i.e., which has side-effect, then we do not get any error or warning message, which makes this possibility very dangerous, because the assertion is only executed if the -enableassert or -ea java option is used.

JML (Java Modeling Language). JML is a behavioral interface specification language [2]. In JML one can specify for each class and interface its behavior.

```
Precondition: requires <predicate>;
Postcondition: ensures <predicate>;
Invariant: invariant <predicate>;
Frame condition: assignable <predicate>;
```

A predicate may contain Java literals, pure method calls, field references, Java operators, and built-in operators like:

```
\forall, \exists, \max, \min, \sum, \result, \old.
```

From the above list we explain only the frame condition. Here can we constraint whether a method may or not change the inner state of the object, and if yes, which fields can be changed by the method. If a method may not change the inner state, i.e., it has no side-effect, then we say that it is a pure method. Pure methods may be called also from JML statements.

Common JML. Lots of tools support JML, but still, it is difficult to use JML in production. The most well known tool is Common JML (also known as Iowa State University JML), see `http://www.eecs.ucf.edu/~leavens/JML-release/JML.html`. It consists of the following main tools:

```
jml: JML syntax checker.
jmlc: Compiles the Java source code and JML specifications.
jmlrac: Runs the class files and checks the JML specification.
```

Unfortunately the development of the Iowa State University JML release is stalled. It supports only Java2 features, so up to version Java 1.4. There are lots of newer tools, like OpenJML or JMLEclipse, but these tools are not very well documented.

ESC/Java2 tool. The most well-know static checker tool for Java is ESC/Java2 [7], where ESC stands for Extended Static Checker. It uses an Automated Theorem Prover (ATP) to check constraints on the source code. The constraints have to be written in JML. If we do not specify any constraint then we still get useful warnings, like possible null dereferences. It supports Java only up to Java 1.4. It does not support Java generic types.

It is quite impressive that ESC/Java2 can detect on compilation time that our program may falsify some of its contracts. To be able to this, ESC/Java2 uses an ATP. It is shipped with the ATP called "Simplify". It can be used by other ATPs, like "z3".

The "Simplify" prover can not validate the following source code, because it cannot decide non-linear arithmetic:

```
public class Main {
    //@ requires n>=0;
    //@ ensures \result > 0;
    public int fact(int n){
        if (n==0) return 1; else return n*fact(n-1);
    }
}
```

Contract4J. Contract4J is a design by contract tool based on Java annotations [4]. It has its own specification language. Some of its language elements are:

```
@Contract - Each class with a contract has to be annotated by this.
@Pre(<predicate>) / @Post(<predicate>) - Pre / post-conditions.
@Invar(<predicate>) - Invariant.
```

Predicates are Java literals, method calls, field references, Java operators, and built-in operators like:

```
 $this, $return, $old, $args[n].
```

It can be utilized as a dynamic checker. The Contract4J tool supports newer Java technologies, but its specification language is not as rich as JML.

Modern Jass. Modern Jass is a design by contract tool that works with the latest versions of Java and is closely related to JML [8]. It uses its own JML like specification language. It is also a dynamic checker, it uses a runtime agent which checks the specifications. Modern Jass supports newer Java technologies, but its specification language is not as rich as JML.

MOP. MOP (Monitoring Oriented Programming) is a programming paradigm built upon runtime verification, in order to develop reliable softwares via monitoring and recovery [5]. MOP takes monitoring as a fundamental software development principle. Java-MOP is a MOP development tool for Java [6]. Its specification processor employs AspectJ for monitor integration. In other words, it translates outputs of logic plugins into AspectJ code, which is then merged within the original program by the AspectJ compiler. It supports currently 6 logic plugins: finite state machines (FSM), extended regular expressions (ERE), context-free grammars (CFG), past time linear temporal logic (PTLTL), future time linear temporal logic (FTLTL), and past time linear temporal logic with calls and returns (PTCaRet).

Since each logic plugin is able to relate multiple runtime states, therefore, MOP is able to describe properties about a sequence of states.

3. Dimensions

3.1. Dimension of expressiveness

One of the dimensions of Java specification languages is the number of states on which they can introduce constraints.

Along this dimension there are three kinds of specification languages. Some specification languages can specify properties about only a single state, others about a pair of states: the state before the method call (input state), and the state after the method call (output state). The most general ones are designed to speak about sequences of states. So the dimension of expressiveness is the following (with specification languages that belong to them):

- properties about a single state (Java assert),
- properties about a pair of states (JML, Contract4j, Modern Jass),
- properties about a sequences of states (MOP).

3.2. Dimension of ability of verification

We can verify our source code by static checking and dynamical checking. Static checking takes place in compilation time. Dynamic checking takes place in runtime.

We can classify the analyzed tools along this dimension in the following way:

- Static checker: ESC/Java2.

- Dynamic checker: Common JML, Contract4j, Modern Jass, MOP-Java.

3.3. Dimension of access program structures

If we want to state a constraint on the source code or type structure, then we have to query the structure of the program. In Java this means that we have to access the class structure and the runtime object structure.

Design patterns [1] are typically static structures. They tell you how should your class hierarchy looks like, which method should be abstract in the ancestor class, and which method calls which one. They can be visualized by UML + textual descript. They can be described also in specification languages as it is done in our first case studies [3]. Such static properties can be checked naturally by static checkers. They could be checked also by dynamic checkers if we can access the static structure, i.e., ask which methods are abstract, and questions like this. Since all cited specification language allow to call pure Java functions from the specification, and since Java has a reflection package, we can ask questions about the static structure using reflection. We show an example in the next section how to do this.

On the other hand, the non-null constraint is a runtime constraint, i.e., it can be checked naturally only at runtime. However, this can be checked also in compilation time using extended static checking techniques like ESC/Java2. Such tools are using an automated theorem prover to check such runtime properties.

So we can give the following table:

Techniques/Goals	**Static checking**	**Dynamic checking**
Source code properties	Natural	Using reflection
Runtime properties	Extended Static Checking	Natural

4. Class Property Check at Runtime Using Reflection

In this section we describe a simple syntactical constraint, which we call "semi-final". A semi-final class can be the super class of other classes, but the children classes may not have a new public method. They can neither override public methods from the super class. They can however have new non-public methods and they can also override them.

A semi-final class is useful for example to create an instance of the template method design pattern [1]. In this design pattern there is a template method, which fixes the steps of an algorithm, i.e., it encapsulates an algorithm. The individual steps are usually abstract method calls in the abstract ancestor class, where the template method is implemented (other methods can be also implemented, but they have not to be public). These abstract methods do not have to be public, since only the template method is meant to be called from other classes. These abstract methods are overridden by the children classes; in this way we can concretize the algorithm.

In Contract4J we can use also a Java expression where we can use a Contract4J expression. Such a Java expression can refer to objects which are accessible at the point where we write our Contract4J statements. Our "trick" is that we use the Java reflection package, which can query runtime information from an object, class, or method. In our case we want to state that a class may not have public methods.

```
@Contract
@Invar("!Tool.hasPublicMethod($this.getClass())")
public abstract class TemplateMethod {...}
class Tool{
public static boolean hasPublicMethod(Class c){
        Method[] ms = c.getDeclaredMethods();
        for(Method m : ms){
            if (Modifier.isPublic(m.getModifiers())) return true;
        }
        return false;
    }
}
```

The method Tool.hasPublicMethod(Class c) returns true if its parameter, the Class c, has no public method.

5. Summary

We have surveyed several specification languages and tools for Java. However this is not a complete survey. For example the following languages and tools are not included here: OCL (Object Constraint Language), UML tools supporting OCL, and other verification tools like KeY, VeriFast, Jahob, Krakatoa, JavaFAN, see `http://verifythus.cost-ic0701.org/included-tools`.

We can summarize that there are a lots of tools supporting behavior specification, still, we are far from the point, where these tools could be used in the everyday work of a software company. It is not easy to understand why Java has no built-in features for behavior specification except the assert statement.

Behavior specification can help to understand the goal, since it is on some higher abstraction level than the source code. So the programmer has to describe his or her program twice, once in imperative style, i.e., in Java, and once in declarative style, i.e., in a behavior specification language like JML, which sounds superfluous. But think on pair programming, two programmers write one code. It sounds also superfluous on first time, but the practice shows that it provides better quality code, and hence, it pays off. We believe that describing the program twice, once in Java and once in JML also would pay off. The only thing that the community needs for this is a JML tool which supports the newest Java version.

References

[1] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., Design Patterns: Elements of Reusable Object-Oriented Software, *Pearson Education*, 1995.

[2] Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P. and Zimmerman, D.M. JML Reference Manual (DRAFT), *http://www.jmlspecs.org/OldReleases/jmlrefman.pdf*, September 2009.

[3] Schreiner, W. A JML Specification of the Design Pattern "Proxy", *Johannes Kepler University, RISC Institute. Technical report*, April 2009.

[4] Wampler, D. Contract4J for Design by Contract in Java: Design Pattern-Like Protocols and Aspect Interfaces *In Proceedings of the Fifth AOSD Workshop on ACP4IS*, pages 27–30, 2006.

[5] Chen, F., Roşu, G., MOP: An Efficient and Generic Runtime Verification Framework. *In Proceedings OOPSLA 2007*, pages 569–588, 2007.

[6] Chen, F., Roşu, G., Java-MOP: A Monitoring Oriented Programming Environment for Java, *Proceedings of TACAS 2005, LNCS-3440*, pages 546-550, 2005.

[7] Rustan, K., Leino, M., Nelson, G., Saxe, J.B. ESC/Java User's Manual, *SRC Technical Note 2000-002*, October 12, 2000.

[8] Rieken, J. Design By Contract for Java - Revised, *Carl von Ossietzky University of Oldenburg, master's thesis*, April 24th, 2007.

Gábor Kusper
e-mail: gkusper@aries.ektf.hu
Hungary, H-3300 Eger, Eszterházy tér 1.

Wolfgang Schreiner
e-mail: Wolfgang.Schreiner@jku.at
Austria, A-4040 Linz, Altenbergerstr. 69.

Gergely Kovásznai
e-mail: kovasz@aries.ektf.hu
Hungary, H-3300 Eger, Eszterházy tér 1.

Gábor Guta
e-mail: Gabor.Guta@jku.at
Austria, A-4040 Linz, Altenbergerstr. 69.

János Sztrik
e-mail: Sztrik.Janos@inf.unideb.hu
Hungary, H-3300 Eger, Eszterházy tér 1.